Abstraction in Modeling and Programming with Associations
Instantiation, Composition and Inheritance
Kristensen, Bent Bruun

Published in:
Advances in Computer Communication and Computational Sciences

DOI:
10.1007/978-981-13-6861-5_2

Publication date:
2019

Document version:
Accepted manuscript

Citation for published version (APA):

Go to publication entry in University of Southern Denmark's Research Portal

Terms of use
This work is brought to you by the University of Southern Denmark.
Unless otherwise specified it has been shared according to the terms for self-archiving.
If no other license is stated, these terms apply:
- You may download this work for personal use only.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying this open access version

If you believe that this document breaches copyright please contact us providing details and we will investigate your claim.
Please direct all enquiries to puresupport@bib.sdu.dk
Abstraction in Modeling and Programming with Associations: Instantiation, Composition and Inheritance

Bent Bruun Kristensen
University of Southern Denmark
bbk@mmmi.sdu.dk

Abstract. The association concept abstracts over collaboration between concurrent autonomous entities. Associations describe collaborations, i.e. requests and coordination of contributions from the entities. And the association concept also supports abstraction in the form of instantiation, composition and inheritance. Simple and expressive abstraction with associations is demonstrated by illustrative examples, schematic language presentations, brief characterizations of qualities, and implementation experiments.

Keywords: Language design; abstraction with associations; instantiation, composition and inheritance; concurrent autonomous entities; qualities.

1 Introduction

The description of collaboration between autonomous entities is typically distributed among the participating entities in terms of their individual contributions, and the coordination is based on synchronization mechanisms. On the contrary, the association concept gives a unified description of collaboration—i.e. both requests and coordination of contributions from entities. The association abstracts over both structural and interaction aspects of the collaboration: It includes roles and a directive, where entities play the roles in interactions and the directive describes interaction sequence between the entities. In [1] the association concept is motivated and defined—and evaluated and related to similar modeling and programming concepts.

The aim of the present paper is to show the potential of abstraction in modeling and programming with instantiation, composition and inheritance of associations. The association offers instantiation i.e. how to create an instance from an association category, composition i.e. how to compose an association category from instances of existing association categories, and inheritance i.e. how to describe an association category by inheriting from an existing association category.

Section 2 is a brief introduction to the association concept, a presentation of a basic example, as well as related work and prerequisites for this kind of abstraction. Section 3 illustrates abstraction in terms of instantiation, composition and inheritance by giving illustrative but complex examples and schematic presentations, and by characterizing the qualities of the abstractions. In section 4 an experimental implementation of abstraction with associations is discussed.
2 Background

2.1 Associations and Autonomous Entities

Associations are inspired from a model for understanding, modeling and programming ambient systems [2]. The association concept offers an alternative way of describing interactions among concurrent autonomous entities. Association \( X \) and entities \( P \) and \( Q \) are illustrated in Fig. 1: Instances of entities \( (E_P \text{ and } E_Q) \) are autonomous (and therefore distinct from ordinary objects/classes [3]), and participate in instance \( A_X \) of association \( X \). Because an entity is autonomous, only the entity itself controls the execution of its methods. An entity has an action part (illustrated by \( \ldots \)) in the form of a sequence of actions (similar to a class in SIMULA [4]). An entity also has list of the actual contributions (i.e. methods to be executed) requested from associations in which an instance of the entity participates. An association requests an instance of an entity to contribute by executing a given method, but the instance itself chooses among the requests. Associations integrate activity (the directive) and role aspects. The directive (illustrated by \( \ldots \)), is an interaction sequence including requests to participating entities. The interactions of the directive are executed sequentially, whereas an entity executes its action part and list of requests interleaved. The description of a role \( (R_P \text{ and } R_Q) \) of an association includes a name of an entity category (like \( P \text{ or } Q \)) that specifies which instances may play the role. The association concept combines activities [5] and roles [6] in one abstraction.

```
entity P {
  void m_P(...) {...}
  ...
  action part {...}
}

entity Q {
  void m_Q(...) {...}
  ...
  action part {...}
}

association X {
  role R_P for P
  role R_Q for Q
  ...
  directive {
    ...
    R_P::m_P(...)...
    ...
    R_Q::m_Q(...)...
  }
}

X: A_X = new X(...) ...
F: E_P = new F(...) ...
Q: E_Q = new Q(...) ...
E_P play A_X, R_P ...
E_Q play A_X, R_Q ...
```

Fig. 2. Textual version of illustration of association and entity.

A textual version of associations and entities is illustrated in the Association Language in Fig. 2, i.e. association \( X \) with instance \( A_X \) and entities \( P \) and \( Q \) with
instances \( E_p \) and \( E_q \). In respect to an autonomous instance \( E_p \) playing role \( R_p \) of \( A_x \), we use the notation \( R_p::m_p() \) for the request of an invocation of the method \( m_p() \) of \( E_p \). But because entities are autonomous the invocation is a request to \( E_p \), i.e. \( E_p \) decides if and when it may execute \( m_p() \) (interleaved with \( E_p \)'s action part). In respect to the directive, different mechanisms for synchronization and communication among entities are possible: In [1] synchronization of autonomous entities is illustrated in rendezvous-based way [7]; In [8] interactions between agents are illustrated by asynchronous message-passing [9].

Concurrent object execution [10] implies that the objects execute in parallel and independently but that these objects at various times synchronize and typically communicate to exchange data. Typically, the language mechanisms used to describe the synchronization are placed at the concurrent objects, i.e. in the action part of entities \( p \) and \( q \) in Fig. 2. On the contrary, by associations the synchronization mechanisms are placed outside the entities and shared by these entities, i.e. in the association \( x \) in Fig. 2. The association is superior with respect to modeling and programming collaboration because it supports our natural understanding of collaborations between autonomous entities and because the concept captures collaboration as a descriptive unit, c.f. association \( x \) in Fig. 2. The association abstraction \( x \) is formed by our conceptualization of collaborations and is essential for understanding, modeling and communication (“Without abstraction we only know that everything is different” [11]): The association is a language concept and not implementation specific technology. As a shared plan for collaboration, the association makes it possible for entities precisely and understandably to explain their ongoing executions, i.e. in Fig. 2 when entity \( E_p \) executes method \( m_p() \) this execution refers to a specific request for example \( R_p::m_p() \) in association \( x \).

2.2 Example with interacting AGV’s

![Fig. 3. Example with interacting AGV’s.](image)

\(^1\) AGV is the acronym for Automatic Guided Vehicle
The examples in the paper are inspired from a project about transportation systems [12]. The project investigates the process of moving boxes from a conveyor belt onto pallets and transporting these pallets. This scenario is inspired from the high bay area of the LEGO® factory with AGV’s, no human intervention and only centralized control. A toy prototype inspired from this system (to bridge the gap between simulation and real physical applications) measures 1.5 by 2m, with three mobile robots (LEGOBots), two input stations, two output stations, a conveyor belt, and a station with empty pallets. The approach supports a fully distributed control for each LEGOBot. A LEGOBot is based on a LEGO® MindstormsTM RCX brick extended with a PDA and wireless LAN. Problems with combining and maintaining the basic technology (including LEGO® MindstormsTM, RCX, PDA, WLAN) has motivated the introduction of a virtual platform.

The basic example is illustrated in Fig. 3. The example includes two AGV’s and two ports, and the objective is to describe how to organize the transport of packages by the AGV’s between the ports. AGV’s A_i and A_o bring packages from port P_i to port P_o (i for input and o for output). AGV A_i brings a package from port P_i to position Here, the AGV’s shift the package from AGV A_i to AGV A_o, and AGV A_o brings the package from position Here to port P_o. The AGV’s move along suitable paths: AGV A_i moves along a1 followed by a2 followed by a3; AGV A_o moves along b1 followed by b2 followed by b3. The AGV’s must synchronize in order to shift the package from A_i to A_o. Furthermore, AGV A_i must synchronize with port P_i to pick up the package, and AGV A_o must synchronize with port P_o to put down the package.

The example presented in Fig. 3 is described in the Association Language in Fig. 4: AGV and Port are entity categories for AGV’s and ports, respectively. Entity Port includes Package x and Position p whereas entity AGV includes Package x, Position Home and method goto(…). Instances of AGV and Port exist at execution time, namely P_i, P_o and A_i, A_o, respectively. The association category Transport is a description of
the interactions of AGV’s and ports. In the instance of Transport, AGV’s A1, A2, and Ports P1, P2 play roles A1, A0, and P1, P0, respectively. The interactions are described by the association Transport on behalf of—and executed by—the participating AGV’s and ports. The synchronization of AGV’s and ports is described specifically by \( P_1::x \rightarrow A_1::x, A_1::x \rightarrow A_2::x \) and \( A_2::x \rightarrow P_2::x \). The clause \((|\ldots,\ldots|)\) means concurrent execution of the (two or more) parts included in the clause \(^2\). The example illustrates synchronization in time and place by “being at the right places before synchronization in time”. In the directive of Transport the meeting point is calculated as Position Here. Next the directive includes the sequence \((|\ldots,\ldots|)\) \( A_1::x \rightarrow A_0::x \) \((|\ldots,\ldots|)\) to be executed sequentially. In the first \((|\ldots,\ldots|)\) clause, the input AGV A1 moves to input Port P1 from which A1 picks up x and moves to Here, and concurrently the output AGV A0 moves to Here. When both A1 and A0 are at Here the package x is transferred from A1 to A0 by \( A_1::x \rightarrow A_0::x \). In the last \((|\ldots,\ldots|)\) clause, the input AGV A1 moves to its Home, and concurrently the output AGV A0 moves to output Port P0, where A0 puts down x and moves to its Home.

The problem outlined in Fig. 3 is illustrative and Fig. 4 offers an excellent solution. Variations of this problem are used in sections 3.2 and 3.3 for illustrating composition and inheritance. The programs are created in order to illustrate the functionality of composition and inheritance but are not straightforward solutions to the problems.

2.3 Related Work and Prerequisites

In [1] the basic form of the association concept is compared to related work—in the present paper, the focus is on instantiation, composition and inheritance by means of associations: In [13] relations are introduced as abstractions between objects. However, the relation is an associative abstraction over structural aspects only, i.e. interaction aspects are not covered by the relation. Alternative approaches in [14], [15], [16] and [17] support the notion of relationships in object-oriented languages. But the purpose is to have structural relationships only—where the essence of the association is also to support interactions through the relationships. The alternative approaches do not support composition and inheritance of relationships—however in [15] the term composition is used to establish something similar to transitivity of relations.

In general, programming languages contain mechanisms with relevant qualities for modeling time, programming time and runtime, e.g. [18], [19] and [20]. Qualities may be discussed either for a specific language, i.e. the whole language or specific mechanisms of a language; Or for languages in general or categories of structures in programming languages, e.g. control structures; Or in the form of specific qualities (e.g. simplicity) where these may be interlinked or overlapping (e.g. simplicity and readability) with the inherent property of often being conflicting (e.g. readability and

\(^2\) \( S::x \rightarrow R::y \) describes synchronization between S and R where S is requested to transfer the value of x, and R to receive this value in y. Entities S and R are synchronized exactly when the contents of x is transferred to y.

\(^3\) \((|\ldots,\ldots,\ldots|)\) describes concurrent execution of the parts included. The execution is completed when all parts have completed their execution.
efficiency). We focus on the development process, i.e. we are concerned only about modeling and programming time, i.e. development time. And we focus on the association concept—where abstraction is a quality itself. In relation to abstraction by means of associations, we focus on qualities of language mechanisms for instantiation, composition and inheritance. For reasons of simplicity we only include the qualities understandability, efficiency and flexibility—with the definition:

— **Understandability**: The clarity and the simplicity of the designed models and described programs. We see simplicity and readability as aspects of understandability.

— **Efficiency**: The effort needed to develop models and programs by using the mechanisms. In general, efficiency may be in conflict with understandability.

— **Flexibility**: The ability to easily change various aspects within the models or programs. In general, flexibility supports both efficiency and understandability.

![Diagram](image)

**Fig. 5.** Illustration of action inheritance in Java (left) and in Beta (right).

To design inheritance for the association concept, action inheritance is essential. In general, action inheritance (e.g. methods in Java and Beta [21] or action sequences in classes in Simula and Beta) differs as exemplified by Java and Beta, as illustrated in Fig. 5 (the syntax is neither from Java nor from Beta): Methods $X$ and $X'$ are declared in classes $Y$ and $Y'$, respectively. Class $Y'$ inherits from class $Y$, whereas method $X'$ inherits from method $X$. Action inheritance in Java is controlled by `super X(...)` in $X'$, where $X'$ is declared as a usual method, and the names $X'$ and $X$ must be identical. Invocation of $X(...)$ of an $Y'$ object gives the sequence $\sim2\sim1\sim3\sim$. The intention is that $\sim1\sim$ is seen as a preliminary sequence to be combined as specified with secondary sequences $\sim2\sim$ and $\sim3\sim$. Action inheritance in Beta is controlled by `inner` in $X$ and by declaring method $X'$ inherits $X (...)$, where the name $X'$ may be different from $X$. The invocation of $X' (...)$ of an $Y'$ object gives the sequence $\sim1\sim3\sim2\sim$. The intention is that $\sim1\sim$ and $\sim2\sim$ are seen as general before and after sequences, respectively, whereas $\sim3\sim$ is a special sequence in between, i.e. a conceptual understanding of action inheritance. Consequently, we use the approach from Beta (and Simula) because the intention of inheritance of associations is to support development of conceptual models.
3 Abstraction

Abstraction in modeling and programming with associations include:
— Instantiation, i.e. an instance of an association is created from the association category,
— Composition, i.e. an association is described as a composition of instances of other associations, and
— Inheritance, i.e. an association is described as an extension of another association.

3.1 Instantiation

Instantiation refers to the creation of an instance from an association category. An association category is a description from which instances are created with unique identities, state and values according to this description. Instantiation supports a number of qualities among which our focus is efficiency and understandability, i.e. we use a single description for creating several instances and our understanding is captured explicitly by the category.

Fig. 1 gives a schematic illustration of instantiation of associations. Fig. 1 (right) illustrates an instantiation of the model in Fig. 1 (left): \( A_x \) is an instantiation of association \( X \), \( E_p \) and \( E_q \) are instances of entities \( P \) and \( Q \), \( E_p \) and \( E_q \) play the roles \( R_p \) and \( R_q \), and an arrow indicates the state of the execution of a directive/an action part.

Instantiation of associations is similar to instantiation in object oriented languages, e.g. Java and Beta, except for the properties role and directive.

In Fig. 6 (left), association \( \text{Transport} \) (with \( \text{Port} \) \( P \), and \( \text{AGV} \)’s \( A_1 \) and \( A_2 \)) is illustrated together with entity categories \( \text{Port} \) and \( \text{AGV} \). This means that \( \text{Transport} \) is identified as an abstraction and described as a category. In Fig. 6 (right), instances of associations and entities are illustrated, namely instance \( T \) of \( \text{Transport} \), instances \( P_1 \), \( P_2 \) of \( \text{Port} \) and \( A_1 \), \( A_2 \) of \( \text{AGV} \), where ports and AGV’s play appropriate roles of \( T \). Fig. 4 shows a textual version of the diagrams in Fig. 6.

For reasons of simplicity we present instantiation of associations only in a very basic form. It is obvious to consider the inclusion of for example constructors also for instantiation of associations.

At development time instantiation of associations especially supports
— Understandability: The purpose of classification is mainly to understand: Without classification, very little is being modelled. By classifying concurrent execution including synchronization and communication by associations, we achieve some understanding and the associations express our understanding.
— Efficiency: Instantiation is time saving because no complete redesign and programming task is necessary—the classification by an association happens only once.

3.2 Composition

Composition refers to the description of an association, the whole association, by combining instances of other associations, the part associations. Composition supports a number of qualities among which our focus is simplicity and understandability, i.e. simple and smaller associations are developed in order for instances of these to be utilized as building blocks and combined in more complicated, but still understandable associations.

In Fig. 7 association Transport with role IO for IOPorts is composed of instances of associations PickUp and PutDown each with roles P for Port and A for AGV. Entity IOPorts is a composition of two Ports. Consequently, the associations PickUp and PutDown are identified as categories where instances of these may be applied in e.g. Transport that also is a category. When applied the instances of PickUp and PutDown are initialized according to their purpose within Transport, i.e. how the AGV’s and ports are used appropriately by PickUp and PutDown.
In Fig. 8 is a textual version of the diagram in Fig. 7. As mentioned, PickUp and PutDown are part associations to pick up and put down a package: In PickUp, AGV A moves to Position p of Port P where the package x is transferred from P to A by P::x → A::x; In PutDown, the package x is transferred from AGV A to Port P by A::x → P::x after which A moves to its Position Home. Entity IOPorts is a composition of two ports P1 and P2 which is used for the IO role in Transport in order to specify the ports from which the transport starts and ends. The allocation of AGV’s Ai and Ao is dynamic e.g. by means of Ai = AGV.allocate(...) the input AGV Ai is allocated. Instances pickup and putdown of PickUp and PutDown respectively, are initialized appropriately concerning Port and AGV. In the remaining directive of Transport the sequence (| ... |) Ai::x → Ao::x (| ... |) is executed sequentially. In the first (| ... |) clause, the execution of the PickUp instance is followed by the input AGV Ai, moving to Here and concurrently the output AGV Ao moves to Here. When both Ai and Ao are at Here the package x is transferred from Ai to Ao by Ai::x → Ao::x. In the last (| ... |) clause, the input AGV Ai moves to its Home and concurrently the output AGV Ao moves to output Port P, followed by the execution of the PutDown instance.
Fig. 9. Diagram illustrating composition.

Fig. 9 is a schematic illustration of composition of associations. The whole association \( X \) is a composed association made up of part associations \( Y_1, \ldots, Y_n \) (a diamond denotes the composition). The part associations \( Y_1, \ldots, Y_n \) are available so that the description of (the properties of) \( X \) may use (the properties of) instantiations of \( Y_1, \ldots, Y_n \), i.e. properties of instances of \( Y_i \) may be used for describing properties of \( X \). In this sense composition of associations is similar to composition in object oriented languages, exemplified by e.g. Java and Beta. The properties of associations, namely roles and directive, cause the difference: Role properties of instances of \( Y_i \) may be used for describing role properties of \( X \), i.e. participants of the part associations are integrated in the description of the participants of the composed association. Similarly, the directive property of instances of \( Y_i \) may be used for describing the directive property of \( X \), i.e. the directives of part associations may be used for executions in the directive of the composed association.

Fig. 10. Schematic illustration of composition for associations.

Fig. 10 shows a textual version as a refinement of the schematic diagram in Fig. 9. Roles and directives of the whole associations may be composed as follows: A role e.g. \( R_p \) or \( R_{p,q_i} \) of the composed association \( X \) is composed of roles of the part associations e.g. \( Y_i, R_{p,i} \) or \( Y_i, R_{p,q_i} \), i.e. roles of instances \( y_i \) are used in the application of the roles of the composed association, e.g. by \( R_p \text{ play } y_i, R_{p,i} \) or by \( R_{p,q_i} \text{ play } y_i, R_{p,q_i} \). The directive of the composed association \( X \) utilizes of directives of instances of parts.
associations as exemplified by \( y_i \), i.e. directives of instances of the parts are included in the description of the directive of the composed association.

For reasons of simplicity we present composition of associations only in a very basic form. Composition is similar to aggregation in [22] where aggregation is used to model whole-part relationships between things. Composite aggregation means that the composite solely owns the part, and that there is an existence and disposition dependency between the part and the composite. Shared aggregation implies that the part may be included simultaneously in several composite instances. It is obvious to consider the inclusion of the distinction between composite and shared aggregation also for composition of associations. Furthermore the distinction in [23] between the *is-component-for* relation and *has-part* relation (not necessarily the reverse relation), may be considered for composition of associations.

At development time composition of associations especially supports

- **Understandability**: The ability to be able to see the whole association as constructed by the various part associations makes the form and content of the whole association more conceivable.
- **Efficiency**: The composition of a whole association from part associations makes the development process simple and straightforward. In this sense the efficiency is based on the simplicity obtained through the divide-and-conquer principle for problem solving in general.

### 3.3 Inheritance

Inheritance refers to the description of an association, the *specialized* association, by supplying additional description to an existing association, the *general* associations. Inheritance supports a number of qualities, among which our focus is *flexibility* and *understandability*, i.e. basic and incomplete associations may be refined to more specific and complete but still reasonably understandable associations.

![Diagram illustrating inheritance in general.](image)

Fig. 11 is a schematic illustration of inheritance in general: The association \( x' \) is a special version of the general association \( x \) (an arrow denotes the inheritance). The general association \( x \) is available so that the description of (the properties of) \( x' \) may use (the properties of) \( x \). In this sense inheritance of associations is similar to inheritance in object oriented languages, exemplified by e.g. Java and Beta. The properties of associations, namely roles and directive, cause the difference: The role properties e.g. \( R_p \) of \( x \) may be used in various ways for describing specialized role properties \( R_p' \) of \( x' \), i.e. participants of the special association are
specialized participants of the general association. Similarly, the directive property of \( x \) may be used for describing the specialized directive property of \( x' \), i.e. the activity of the special association is the activity of the general association including additional interaction sequences.

Fig. 12. Inheritance of PickUpPutDown: inner (left) and virtual associations (right).

Fig. 12 shows diagrams illustrating two variants of directive inheritance as a refinement of the general inheritance from Fig. 11: The leftmost part is based on the inner mechanism whereas the rightmost part is based on virtual associations—also shown in textual form in Fig. 13 and Fig. 15, respectively.

3.3.1 Inheritance by means of Inner

In the illustration Fig. 12 (left) based on inner, the general association PickUpPutDown is used by the specialized association Transport. PickUpPutDown uses instances pickup and putdown of associations PickUp and PutDown (from Fig. 8) in order (before execution of inner) to pick up a package and (after execution of inner) to put down a package. The directive(...) of Transport describes the interactions to replace inner, i.e. to take place between these executions of pickup and putdown.
In Fig. 13 the associations `PickUpPutDown` and `Transport` are given in textual form, corresponding to diagram Fig. 12 (left). As mentioned, `PickUpPutDown` uses instances of associations `PickUp` and `PutDown`, and allocates appropriate AGV’s to be used. In the directive of `PickUpPutDown`, instances of `PickUp` and `PutDown` are initialized appropriately by ports `Pi`, `Po`, and AGV’s `Ai`, `Ao`, i.e. `Ai`, `Pi` in instance `pickup` and `Ao`, `Po` in instance `putdown`. The remaining part of the directive has the form `pickup inner putdown`: Before `inner`, input AGV picks up the package and after `inner`, the output AGV puts down the package. In `Transport` the contents of `inner` is added: The meeting point is calculated as `Here` after which the AGV’s move to the meeting point, exchange the package and move to appropriate places by the sequential sequence `(| ... |) Ai::x → A0::x (| ... |)`. In the first `(| ... |)` clause, the input AGV `Ai` moves to `Here` and concurrently the output AGV `A0` moves to `Here`. When both `Ai` and `A0` are at `Here` the package `x` is transferred from `Ai` to `A0` by `Ai::x → A0::x`. In the last `(| ... |)` clause, the input AGV `Ai` moves to its `Home` and concurrently the output AGV `A0` moves to Port `Po`.

**Fig. 14.** Schematic illustration of inheritance for associations: `inner`.

```
association X { role R for P ~1~ inner ~2~ }  association X' extends X { role R for Q' ~1~ inner ~2~ }  entity P { ... }  entity Q' extends Q { ... }
association X { role R for P ~1~ inner ~2~ }  association X' extends X { role R for Q' ~1~ inner ~2~ }  entity P { ... }  entity Q' extends Q { ... }
association X { role R for P ~1~ inner ~2~ }  association X' extends X { role R for Q' ~1~ inner ~2~ }  entity P { ... }  entity Q' extends Q { ... }
```
Fig. 14 (left) shows a refinement of the schematic diagram in Fig. 11 based on inner, and Fig. 14 (right) gives a textual version: Roles of the general association \( X \) may be inherited by the special association \( X' \) as follows: Any role is inherited from the general association e.g. \( R_1 \) and additional roles may be described for the specialized association e.g. \( R_2 \). An existing role may be modified, i.e. role \( R_{q1} \) from role \( R_q \) in \( X \) may be specialized by replacing \( Q \) by \( Q' \) (where \( Q' \) inherits \( Q \)), i.e. in of \( X' \) role \( R_{q2} \) may be modified as role \( R_q \) for \( Q' \).

The directive is always inherited from the general association and may be modified (the inherited directive may be extended by a supplementary description), i.e. inheritance of the directive is similar to the inner mechanism [4]: In Fig. 14 (right) in the directive of \( X \) the notation ~1~ (indicating some initial general interaction sequence) is followed by inner that is followed by the notation ~2~ (indicating some concluding general interaction sequence). The meaning of inner is that the directive in any association \( X' \) inheriting from \( X \) takes the place of inner in instances of \( X' \): The supplementary description in \( X' \) is indicated by the notation ~3~. The modified directive of an instance of \( X' \) is the sequence ~1~ ~3~ ~2~. An alternative association \( X'' \) also inheriting from \( X \) may describe another directive to replace ~3~.

```plaintext
association PickUp { 
  role P for vP 
  role A for vA 
  vP :< Port, vA :< AGV 
  directive { 
    A::goto(P.p) 
    P::x ~ A::x 
    inner 
  } 
} 

association PickUp2 
  extends PickUp 
  vP :< Port2, vA :< AGV2 
  directive { 
    P::x ~ A::y 
  } 

association PutDown { 
  role P for vP 
  role A for vA 
  vP :< Port, vA :< AGV 
  directive { 
    A::x ~ P::x 
    inner 
    A::goto(A.Home) 
  } 
} 

association PutDown2 
  extends PutDown 
  vP :< Port2, vA :< AGV2 
  directive { 
    A::y ~ P::y 
  } 

association PickUpPutDown { 
  role P_i for vP 
  role P_s for vP 
  association InitPUPD { 
    directive { 
      P_i play vpickup.P 
      P_s play vputdown.P 
      inner 
    } 
  } 
} 

association Transport 
  extends PickUpPutDown { 
    association InitT2 
      extends InitPUPD { 
        directive { 
          A2 play vpickup.A 
          A2 play vputdown.A 
        } 
      } 
  } 

vP :< Port 

Transport T = new Transport() 
Port P = new Port2() 
P1 = new Port2() 
P2 = new Port2() 
P1 play T.P, P2 play T.P
```

Fig. 15. Textual form of inheritance of PickUpPutDown: Virtual associations.
### 3.3.2 Inheritance by means of Virtual Associations

In the illustration Fig. 12 (right) based on virtual associations, the association PickUpPutDown is specialized to the association Transport. The intention of Transport is to move two packages (instead of one package) between the ports. Consequently, entity Port is specialized to Port2 in order also to include package y (entity AGV is specialized to AGV2 similarly). Associations PickUp and PutDown are both extended to include inner in order to allow the interactions not only to pick up and put down x but to be specialized in order also to pick up and put down y. Associations PickUp2 and PutDown2 describe this inheritance from PickUp and PutDown. Consequently, the directive of association PickUpPutDown is described by means of instances of virtual versions of associations PickUp and PutDown, namely the virtual associations vPU and vPD, respectively. These associations are declared virtual by vPU:<PickUp and vPD:<PutDown, respectively. The actual instances vpickup and vputdown are declared accordingly, and the sequence vpickup inner vputdown specifies execution of vpickup, before execution of inner, before execution of vputdown. In Transport the declarations of the virtual associations vPU and vPD are further specialized to be exactly of categories PickUp2 and PutDown2 in instances of Transport by the declarations vPU:<PickUp2 and vPD:<PutDown2, respectively. Consequently, directive{...} of Transport describes the interactions to be executed between the executions of these specialized versions of vpickup and vputdown.

In Fig. 15 the associations PickUpPutDown and Transport are given in textual form, corresponding to diagram Fig. 12 (right). Therefore, both Port and AGV are specialized to Ports2 and AGV2, respectively. Port, Port2 and AGV, AGV2 describe entities and are used in the associations to declare virtual entities. This means that the declarations vP:<Port and vA:<AGV in PickUp and PutDown are re-declared as vP:<Port2 and vA:<AGV2 in PickUp2 and PutDown2, respectively. Consequently, the instances of the entities declared by means of vP and vA are specialized to be declared by Port2 and AGV2, respectively. Another change of the purpose of the example is that only a single AGV2 picks up and puts down the packages instead of having two AGV’s to meet in order to exchange a package. Consequently, only a single AGV2 is allocated in Transport and this single AGV2 (named A2) is the only AGV used in the example. In order to be able to specialize the associations PickUp and PutDown to pick up and put down not only one package but two packages, the declarations of the associations are (as mentioned) specialized by means of the inner mechanism. The associations PickUp2 and PutDown2 specialize PickUp and PutDown appropriately to also pick up and put down y.

In the association PickUpPutDown, the virtual associations vPU and vPD are declared as well as instances vpickup and vputdown of these. In association Transport, the virtual associations vPU and vPD are specialized to PickUp2 and PutDown2, implying that in an instance of Transport, the instances vpickup and vputdown actually are instances of PickUp2 and PutDown2, respectively. The sequence vpickup inner vputdown in the directive of PickUpPutDown is utilized in Transport where its directive replaces inner in this sequence. Consequently, before inner, x and y are picked up by A2—and after inner, x and y are put down by A2. In between, the
directive of Transport, namely $A2\::\text{goto}(P_o.p)$, replaces \textit{inner}, meaning that in between $A2$ moves from the input $Port$ to the output $Port$.

Furthermore, association $\text{initPUPD}$ is a local association to $\text{PickUpPutDown}$ used as an initialization in the beginning of the directive of $\text{PickUpPutDown}$ to assign which ports to play which roles. In association Transport the association $\text{initPUPD}$ is specialized to $\text{initT2}$ to also assign which AGV’s play which roles, in order also to take care of this initialization in instances of Transport.

![Diagram of inheritance for associations: Virtual associations.](image)

Fig. 16. Schematic illustration of inheritance for associations: Virtual associations.

Fig. 16 (left) shows a refinement of the schematic diagram in Fig. 11 based on virtual associations, and Fig. 16 (right) gives a textual version: Instance $iV$ is declared in the general association $\mathcal{Y}$ to be an instance of virtual association $\mathcal{V}$ that is declared to be an association of category $\mathcal{X}$ (by virtual declaration $\mathcal{V}<\mathcal{X}$). Instance $iV$ becomes an instance of $\mathcal{X}'$ (a specialized association of $\mathcal{X}$) when the virtual association $\mathcal{V}$ is further specialized to $\mathcal{X}'$ in the specialized association $\mathcal{Y}'$ (by the virtual re-declaration $\mathcal{V}<\mathcal{X}'$). The effect is that the instance $iV$ of category $\mathcal{X}$ in instances of $\mathcal{Y}$ retroactively becomes of category $\mathcal{X}'$ in instances of $\mathcal{Y}'$ despite that instance $iV$ actually is declared in $\mathcal{Y}$. The virtual association is similar to a virtual class in Beta [24] and [25]: The virtual association inherited may be further specialized by succeeding re-declarations.

For reasons of simplicity we present inheritance of associations only in a basic form. Multiple inheritance refers to a feature of some object-oriented programming languages in which a class can inherit behaviors and features from more than one superclass. Multiple inheritance for the association concept implies almost identical complications, and instead we recommend to support multiple inheritance for associations similar to the approach based on part objects in [21].

### 3.3.3 Characterization: Inheritance of Associations

Inheritance, c.f. the previous sections, is seen as \textit{structural parameterization} (by descriptions, not only by variables and values). The \textit{inner} mechanism offers a fixed form, whereas virtual associations offer a generalized version.
At development time inheritance of associations especially supports

— Flexibility: Parameterization supports flexibility: Inheritance of associations is structural parameterization, i.e. at development time inheritance enables description of additional associations by parameterizing existing associations.

— Understandability: Given that associations are related by inheritance, the model or program becomes easier to comprehend due to the underlying conceptual model. Furthermore, the model or program typically becomes smaller but more complex and thus typically less understandable.

4 Experimental Implementation

The purpose of the experimental implementation is to test the informal definition of the Association Language, to experiment with the execution of example programs written in this language, and to obtain experience with translator and runtime system. The implementation includes a translator and a virtual environment (implemented in Java). The translator transforms programs in the Association Language (mainly association and entity abstractions) to Java in terms of predefined abstract classes Association and Entity. The translated classes are included in the virtual environment, in which an interpreter produces an execution. The virtual environment is illustrated in Fig. 17: The rightmost part is a visualization of the system. The leftmost part is a logical framework with abstract classes Entity, Association and Context. The interpreter system in the middle part maintains the repository of instances of entities and associations. When entities and activities are interpreted the environment visualizes AGV’s, ports and packages.

The implementation is experimental in the sense that neither completeness nor (translation and runtime) efficiency are important. The translation from the Association Language to Java is supported by a simple translation scheme [27]. The translation is incomplete in the sense that ordinary parts of the Java programming language (included in the Association Language) are not checked. Furthermore the contextual analysis [28] is completed only when necessary, e.g. declarations and applications of associations and entities are matched, instantiated associations and entities are checked to be declared, etc.—by means of a simple symbol table [27]. Support of local associations and extensive analysis of declaration and re-declaration of virtual associations are incomplete parts of the translation. The interpreter part [28]

---

4 An ambient system may also include contexts, i.e. universes in which entities and associations exist. In [26] contexts are named habitats and conceived as some kind of locality, providing to its inhabitants opportunities that allow its inhabitants to achieve their various goals.
is experimental in the sense that e.g. composition of associations is implemented by
copying the directive of a part association into the directive of a whole association
(and by substituting variables according to the ... play ... specifications). Similarly,
the combination of directives by means of inheritance of associations is implemented
by mixing the directive of the special association and the directive of the general
association by replacing the inner clause appropriately.

Fig. 18. Snapshot of BotLife System and BotLife Visualization.

The execution is presented by the BotLife System where the behavior of AGV’s,
ports and packages is illustrated, cf. Fig. 18: The two white spots numbered 0 and 1
show the input AGV and the output AGV. The illustration matches the situation in Fig. 3:
The input AGV moves from its Home towards the input Port and the output AGV moves
from its Home to the meeting point Here. The Ports are loaded with packages
indicated by small boxes. The input AGV picks up a package at the input Port and
brings it to Here. The output AGV takes over the package, brings it to the output Port
and puts down the package.

Fig. 18 illustrates a general visualization of associations and entities exemplified
by Transport and AGV 0, AGV 1, Port 2, Port 3, respectively. The internal situation
(i.e. relation to the program description) in the BotLife Visualization (Fig. 18
right) matches the external situation in the BotLife System (Fig. 18 left). The column
below each of the entities AGV 0, AGV 1, Port 2 and Port 3 shows the actual
requested contributions from the entity. The method currently being executed by an
entity is in white whereas the remaining pending methods are in black (in the actual
snapshot only a white method is present) and the number in front of the method refers
to the (number of) the requesting association (in this case 1 corresponding to
Transport). The column below Transport shows its directive, i.e. its static
operations. The clause (| ... |) is white because it is being executed. And because the
operands of this clause are being executed concurrently, an operation in each of the
two sequences of the clause is also in white, namely the operations corresponding to
A::goto(P.p) and A::goto(Here). For reasons of simplicity of the
implementation, the actual parameters are replaced by ... and the names of the AGV’s
and ports are replaced by a unique numbering of entities, namely 0, 1, 2, and 3,
corresponding to AGV 0, AGV 1, Port 2 and Port 3, respectively. The BotLife
System offers an instructive dynamic visualization of the semantics of the mechanisms of the Association Language.

5 Conclusions

The use of associations for abstraction in the form of instantiation, composition and inheritance supports our conceptualization of collaborations and is essential for understanding, modeling and communication about collaborations. Also these abstractions are simple and expressive at software development time: Instantiation and composition are characterized by understandability and efficiency, whereas inheritance is characterized by understandability and flexibility.

Future work includes additional experiments with software development based on instantiation, composition and inheritance for associations, as well as with efficiency of the implementation.

References


